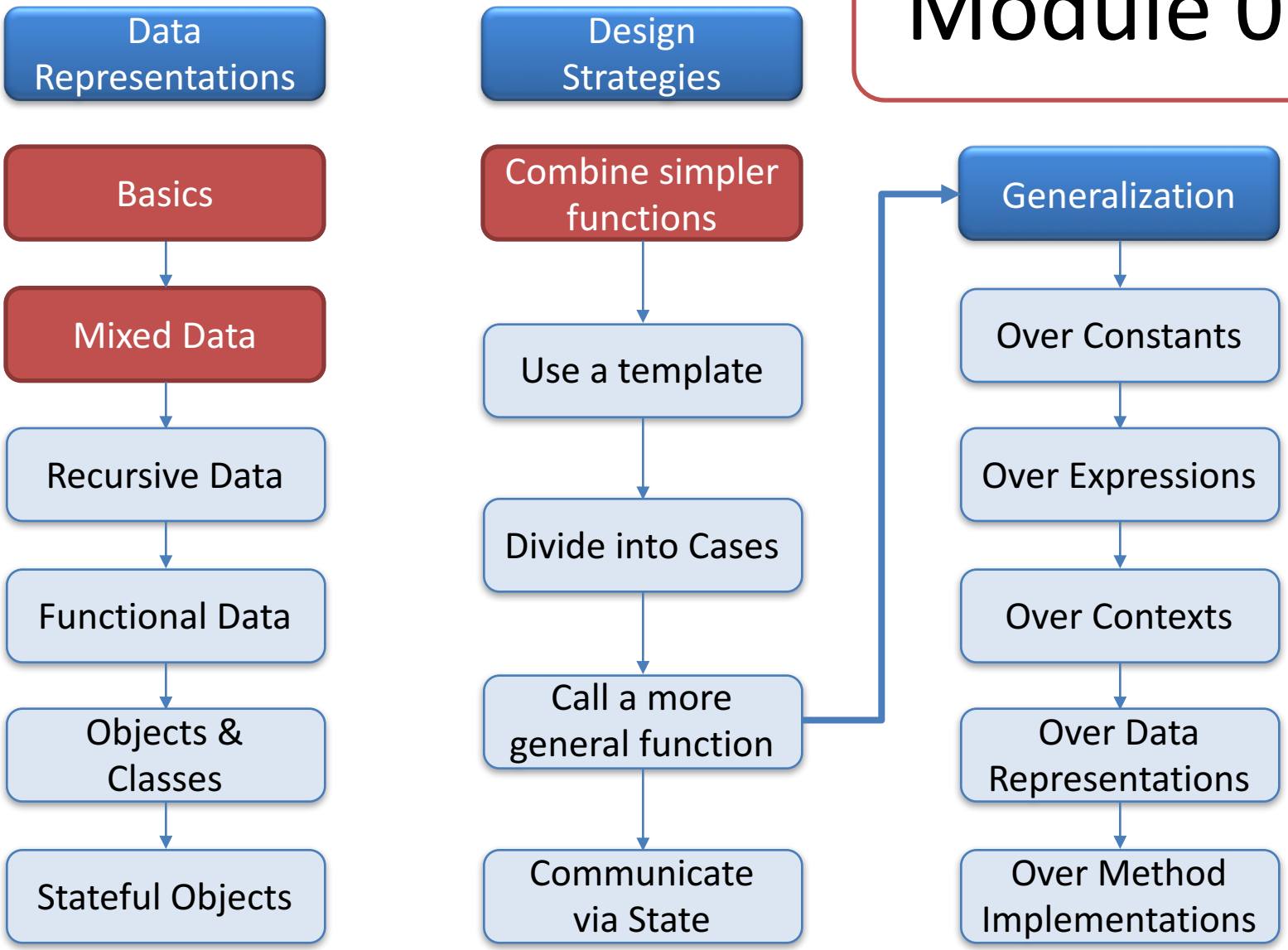
# The Function Design Recipe

CS 5010 Program Design Paradigms

Lesson 1.1

# Module 01


Data Representations
Basics
Mixed Data
Recursive Data
Functional Data
Objects & Classes
Stateful Objects
Design Strategies
Combine simpler functions
Use a template
Divide into Cases
Call a more general function
Communicate via State
Generalization
Over Constants
Over Expressions
Over Contexts
Over Data Representations
Over Method Implementations

# Learning Objectives

- By the time you complete this lesson, you should be able to:
  - list the 6 steps of the Function Design Recipe
  - briefly explain what each step is.
  - explain the difference between information and data, and explain the role of representation and interpretation.

# The function design recipe

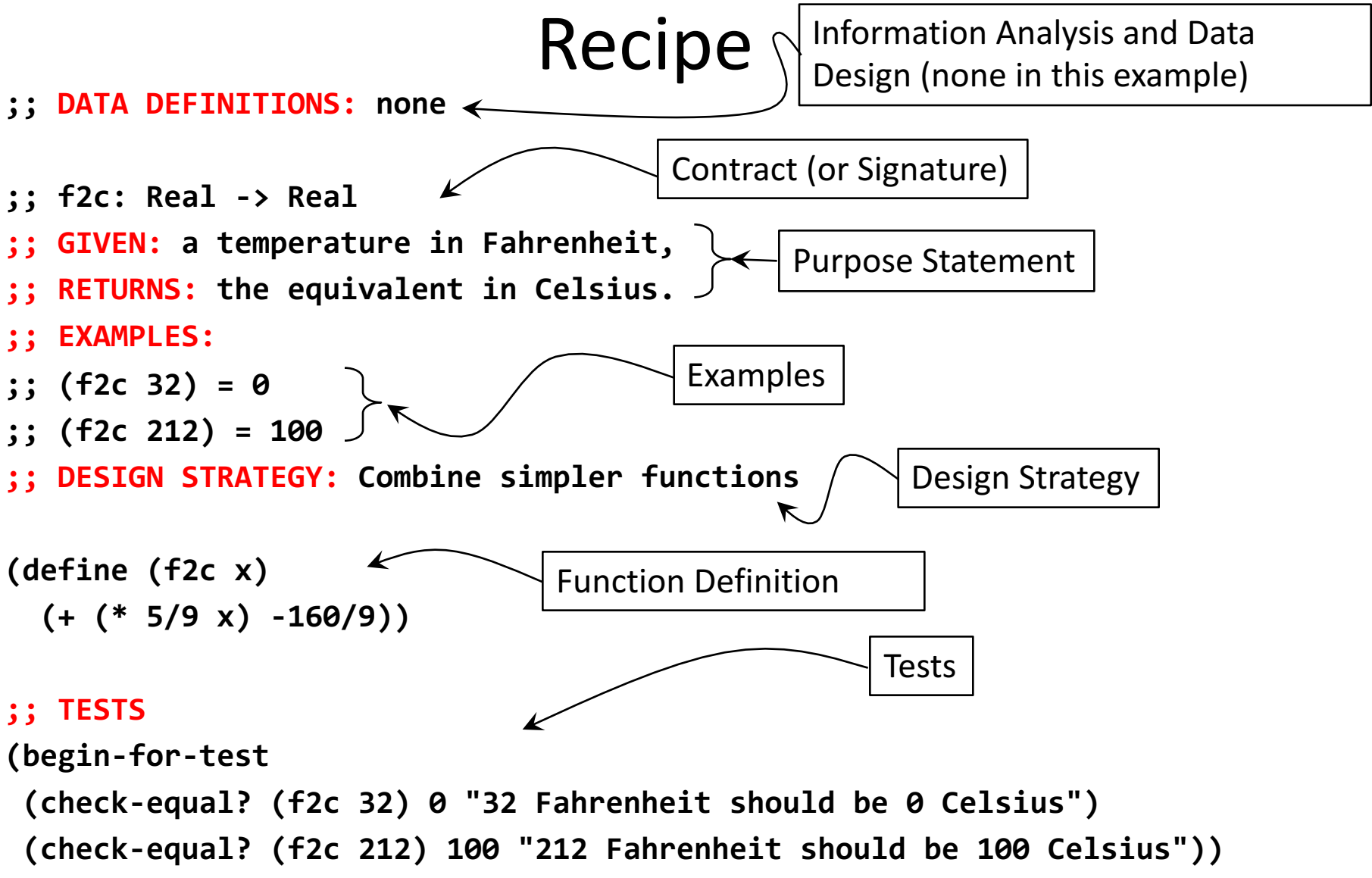
- The function design recipe is the most important thing in this course. It is the basis for everything we do.
- It will give you a framework for attacking any programming problem, in any language. Indeed, students have reported that they have found it useful in other courses, and even in their everyday life.
- With the recipe, you need never stare at an empty sheet of paper again.
- Here it is:

# The Function Design Recipe

| The Function Design Recipe |
| --- |
| 1. Data Design |
| 2. Contract and Purpose Statement |
| 3. Examples and Tests |
| 4. Design Strategy |
| 5. Function Definition |
| 6. Program Review |

*This is important. Write it down, in your own handwriting. Keep it with you at all times. Put it on your mirror. Put it under your pillow. I'm not kidding!*

# A Function Designed According to the Recipe



# The recipe is a recipe

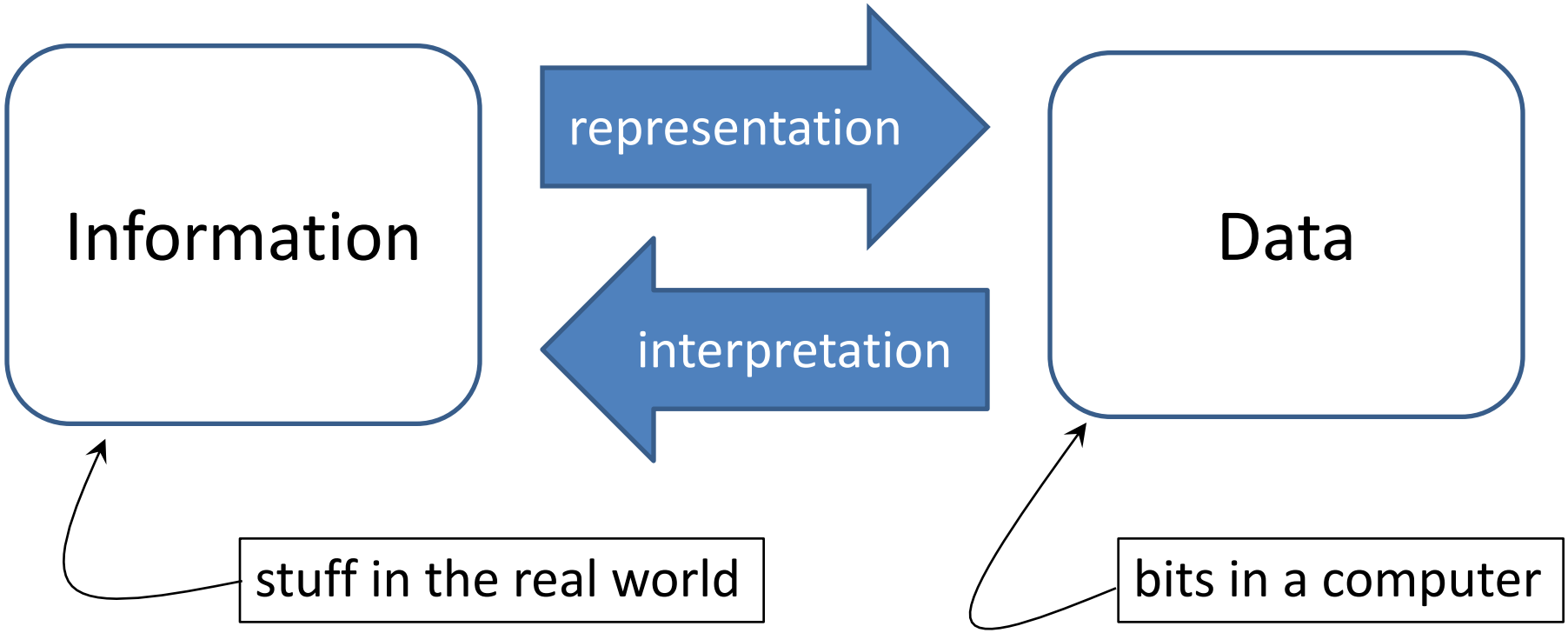
- It's not just a list of components
- It tells you the *order* in which you should do them.
- Each step depends on the preceding ones.
- If you do them out of order, you *will* get in trouble (trust me!)

*In the rest of this lesson, we will discuss each step in turn, illustrating them using our f2c example.*

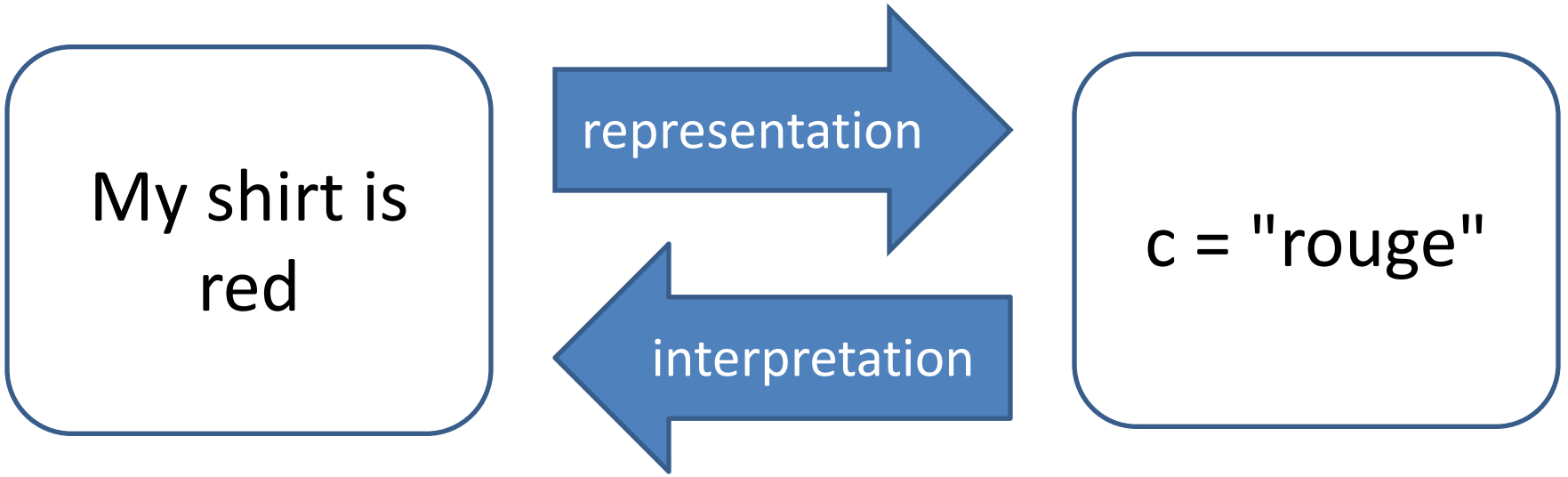
# Step 1: Information Analysis and Data Design

- Information is what lives in the real world. To do this step, you need to do 3 things:
  1. You need to decide *what part* of that information needs to be represented as data.
  2. You need to decide *how* that information will be represented as data
  3. You need to document how to *interpret* the data as information

# The relation between information and data



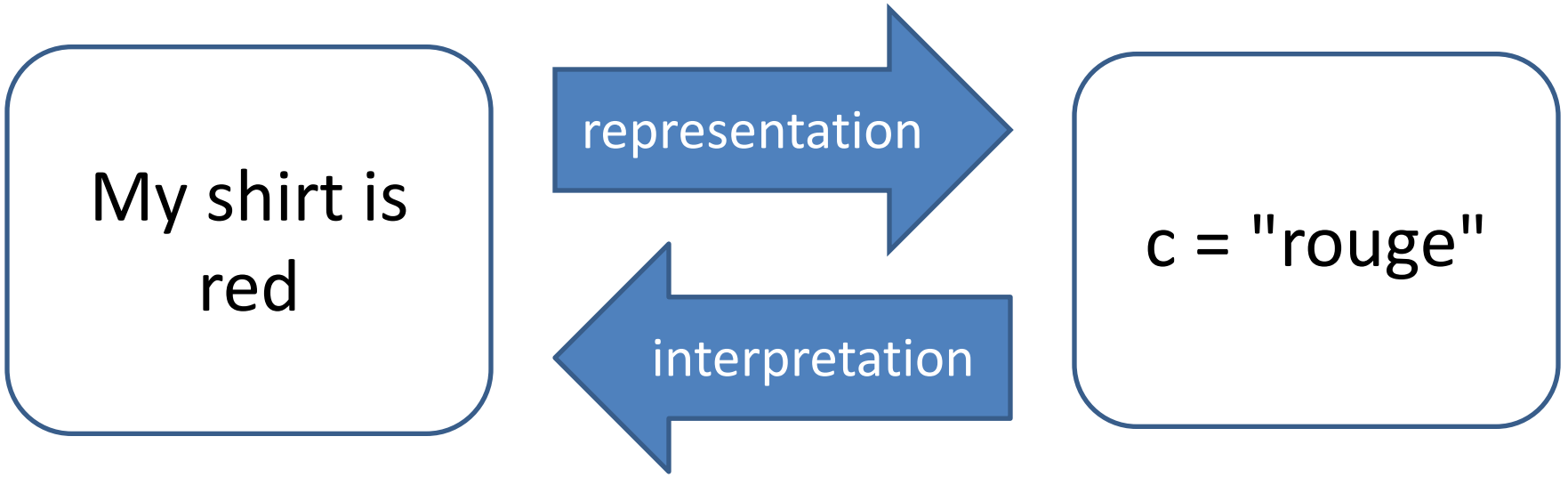
# Information and Data: Example




How do we know that these are connected?

Answer: we have to write down the interpretation.

# Information and Data: Example




Interpretation:

c = the color of my shirt, as a string, in French

This is part of the program *design* process.

# Deliverables for Step 1 (Information Analysis and Data Design)

- Constructor Template
  - a recipe for how to build values of this data type
- Observer Template
  - a template for functions that look at values of this type
- Interpretation
  - what each value of the type represents
- Struct Definitions
  - declarations of new data structures, if any
- Examples
  - see Slogan #2 (Lesson 0.2)

# That first step was a big one!

- ... but important:  the vast majority of errors in student programs can be traced back to errors in step 1!

# Step 2: Contract and Purpose Statement

- *Contract:* specifies the kind of input data and the kind of output data
- *Purpose Statement:* A set of short noun phrases describing *what* the function is supposed to return. These are typically phrased in terms of information, not data.
  - The generally take the form GIVEN/RETURNS, where each of these keywords is followed by a short noun phrase.
  - When possible, they are phrased in terms of information, not data.

# Examples of Contract and Purpose Statements


```
f2c : Real -> Real
GIVEN: a temperature in Fahrenheit
RETURNS: the corresponding temperature in
Celsius

add-cat-to-scene : Cat Scene -> Scene
GIVEN: a Cat c and a Scene s
RETURNS: A Scene like s, except that the Cat c
has been painted on it.
```

# A Cat is not a cat

We wrote:

```
add-cat-to-scene : Cat Scene -> Scene
GIVEN: a Cat c and a Scene s
RETURNS: A Scene like s, except that the Cat c has been
  painted on it.
```

but of course there are no cats in our computer. What this means is:

- **c** is the representation of some cat
- **s** is the representation of some scene
- the function returns a representation of a scene like the one **s** represents, except the new scene contains an image of the cat.

# Step 3: Examples

- Some sample arguments and results, to make clear what is intended.

```
(f2c 32) = 0
(f2c 212) = 100
```

# Tests

We will use the **`rackunit`** testing framework. Your tests will live in the file with your code, so they will be run every time you load your file. That way if you inadvertently break something, you'll find out about it quickly. More on this later.

```
(check-equal? (f2c 32) 0
  "32 Fahrenheit should be 0 Celsius")
(check-equal? (f2c 212) 100
  "212 Fahrenheit should be 100 Celsius")
```

# Step 4: Design Strategy

- A short description of how to get from the purpose statement to the function definition
- We will have a menu of strategies.
- We'll cover this in more detail in Module 2

Here is our starting list of strategies:

There will be more…

| Design Strategies |
| --- |
| 1. Combine simpler functions |
| 2. Use template for <data def> |
| 3. Divide into cases on <condition> |

# Design Strategy for f2c

- For f2c, the strategy we will use is “combine simpler functions”
  - this is, we'll just assemble our function from functions we already have on hand

# Step 5: Function Definition

To define our function, we apply some external knowledge. We know that Fahrenheit and Celsius are related linearly, so the solution must be of the form

$$f2c(x) = ax + b.$$

So we take our two examples and get two simultaneous equations:

$$x = 0: \quad 32a + b = 0$$
$$x = 212: \quad 212a + b = 100$$

We solve for a and b, getting

$$a = \frac{5}{9}, b = -\frac{160}{9}$$

# Function Definition

- Now we can write the code.
  - Our code is just a transcription of the formula into Racket, using the fact that Racket has rational numbers.

```
(define (f2c x)
  (+ (* 5/9 x) -160/9))
```

# Step 6: Program Review

- Did the tests pass?
- Are the contracts accurate?
- Are the purpose statements accurate?
- Can the code be improved?

# Summary

- In this lesson, we have learned the steps of the Function Design Recipe.
  - 6 steps
  - Gives a plan for attacking any programming problem
  - The single most important thing in this course!!

# Next Steps

- Review 01-1-f2c.rkt in the Examples folder.
  - Download and run it. Make some changes. What happens when you change the file?
- If you have questions about this lesson, post them on Piazza.
- Go on to the next lesson.